Component-based Software

Introduction and overview

Motivation for Component Based Development

- Divide-and-conquer (Alexander the Great)
- Well known in other disciplines
  - Mechanical engineering (e.g., German DIN 2221; IEEE standards)
  - Electrical engineering
  - Architecture
  - Computer architecture
- Outsourcing to component producers
  (components off the shelf, COTS)
- Goal: Reuse of partial solutions
- Easy configurability of the systems
  - Variants, versions, product families

Recommended Reading

- [ISC] Chapters 1 + 2
- Douglas McIlroy's home page
  http://cm.bell-labs.com/who/doug/
  http://cm.bell-labs.com/cm/cs/who/doug/components.txt

Mass-produced Software Components

- Garmisch 1968, NATO conference on software engineering
- McIlroy:
  - Every ripe industry is based on components, since these allow to manage large systems
  - Components should be produced in masses and composed to systems afterwards
in the phrase "mass production techniques," my emphasis is on "techniques" and not on mass production plain. Of course, mass production, in the sense of limitless replication of a prototype, is trivial for software.

Yet this fragile analogy is belied when we seek for analogues of other tangible symbols of mass production.

- The idea of subassemblies carries over directly and is well exploited.
- The idea of interchangeable parts corresponds roughly to our term "modularity," and is fitfully respected.
- The idea of machine tools has an analogue in assembly programs and compilers.

Later McIlroy was with Bell Labs ... and invented pipes, diff, join, echo (UNIX).

Pipes are still today the most employed component system!

Where are we today?
Definitions of Components (cont.)

- Heineman / Councill [Ch.1]:
  “A software component is a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard.
  A component model defines specific interaction and composition standards.
  Composition is the combination of two or more software components yielding a new component behavior at a different level of abstraction ... [which is] determined by the components being combined and the way how they are combined.”

Real Component Systems

- Lego
- Square stones
- Building plans
- IC’s
- Hardware bus
- How do they differ from software?

What is a Component? [ISC/CS]

- A component is a container with
  - variation points
  - extension points
  - that are adapted during composition
- A component is a reusable unit for composition
- A component underlies a component model
  - abstraction level
  - composition time (static or runtime?)

What Is A Component-Based System?

- A component-based system has the following divide-and-conquer feature:
- A component-based system is a system in which a major relationship between the components is
  - tree-shaped
  - or reducible.
- Consequence: the entire system can be reduced to one abstract node
  - at least along the structuring relationship
- Systems with layered relations (dag-like relations) are not necessarily component-based.
  - Because they cannot be reduced
What Is A Component-Based System?

- Because it is divide-and-conquer, component-based development is attractive.
- However, we have to choose the structuring relation.
- And, we have to choose the composition model.
- Mainly, two sorts are known:
  - Modular decomposition (blackbox)
  - Separations of concerns (graybox)

Component Systems (Component Platforms)

- We call a technology in which component-based systems can be produced a component system or component platform.
- A component system has

Component Models

Composition Techniques

Component System Composition Technique

for description of components

for compositions of components

Software Component Composition Systems

- A composition system has

Component Model

Composition Technique

Composition Language

for programming-in-the-large and architecture

The Ladder of Component and Composition Systems

<table>
<thead>
<tr>
<th>Architecture Systems</th>
<th>View Systems</th>
<th>Software Composition Systems</th>
</tr>
</thead>
<tbody>
<tr>
<td>Classical Component Systems</td>
<td>Standard Components</td>
<td>.NET CORBA Beans EJB</td>
</tr>
<tr>
<td>Object-Oriented Systems</td>
<td>Objects as Run-Time Components</td>
<td>Java</td>
</tr>
<tr>
<td>Modular Systems</td>
<td>Modules as Compile-Time Components</td>
<td>Ada-85</td>
</tr>
</tbody>
</table>

Component Systems

Component Platforms

Component Model

Composition Technique

Component System

Composition Technique

for composition of components
Desiderata for Flexible Software Composition

- Component Model:
  - How do components look like?
  - Secrets, interfaces, substitutability
- Composition Technique
  - How are components plugged together, composed, merged, applied?
  - Composition time (Deployment, Connection, ...)
- Composition Language
  - How are compositions of large systems described?
  - How are system builds managed?
- Be aware: This list is NOT complete!

The Essence of the 60s-90s: LEGO Software

- Procedural systems
- Modular systems
- Object-oriented technology
- Component-based programming
  - CORBA, EJB, DCOM, COM+, .NET
- Architecture languages

Blackbox Composition

- Procedure Systems
  - Fortran, Algol, Pascal, C, ...
  - The procedure is the static component
  - The activation record the dynamic one
  - Component model is supported by almost all processors directly
    - JumpSubroutine instruction
    - Return instruction
Procedures as Composition System

- **Component Model**
  - Content: binary code with symbols
  - Binding points: linker symbols
  - Procedures (with parameters) and global variables

- **Composition Technique**
  - Connection by linking object files
  - Program transformation on object files
  - Composition time: link-time, static

---

Modules (a la Parnas)

- We can attempt to define our modules "around" assumptions which are likely to change. One then designs a module which "hides" or contains each one.
- Such modules have rather abstract interfaces, which are relatively unlikely to change.
- Every module hides an important design decision behind a well-defined interface which does not change when the decision changes.

---

Modules

- Static binding of functional interfaces to each other
- Concept has penetrated almost all programming languages (Modula, Ada, Java, C++, Standard ML, C#)

---

A Linker is a Composition Operator That Composes Modules

- Provided
- Required

- Bound procedure symbols, no glue code
Modules as Composition System

**Component Model**
- Content: groups of procedures
- Binding points: linker symbols
- Procedures (with parameters) and global variables

**Composition Technique**
- Connection by linking object files
- Program transformation on object files
- Composition time: link-time, static

Unix Filters and Pipes as Composition System

**Component Model**
- Content: unknown (due to parsing)
- External bytes
- Binding points: stdin/out ports
- Secrets: distribution, parallelism

**Composition Technique**
- Adaptation: filter around other components
- Filter languages such as sed, awk, perl
- Binding time: static

Object-Oriented Systems

**Components**
- Objects: objects (runtime) and classes (compile time)
- Objects are instances of classes (modules) with unique identity
- Objects have runtime state
- Late binding of calls by search/dispatch at runtime

UNIX Filters and Pipes [McIlroy]

- UNIX shells style still offers the most used component paradigm:
  - Communication with byte streams via standard I/O ports
  - Parsing and linearizing the objects
  - Extremely flexible, simple
**Object-Oriented as Composition System**

**Component Model**
- Content: binary files, objects (code and data)
- Binding points: static (monomorphic) and polymorphic (dynamically dispatched) calls

**Composition Technique**
- Adaptation by inheritance or delegation
- Extensibility by subclassing

**Composition Language**

---

**Commercial Component Systems**

- CORBA / DCOM / .NET / JavaBeans / EJB
- Although different on the first sight, turn out to be rather similar

**CORBA**
- http://www.omg.org/corba
- Language independent, distribution transparent
- Interface definition language IDL
- Source code or binary

**Interface Specification (in IDL)***
- Generate

---

**COM, ActiveX**
- http://www.activex.org
- Microsoft's model is similar to CORBA. Proprietary
- (D)COM is a binary standard

---

**Software Bus**

---

**Mediator**

---

**Clients, Registry**

---

**Objects**

---
Java Beans

- Java only: source code / bytecode-based
- Event-based, transparent distribution by remote method invocation (RMI – includes Java Object Serialization)

DOT-NET

- Language independent, distribution transparent
- NO interface definition language IDL (at least for C#)
- Source code or bytecode MSIL
- Common Language Runtime CLR

CORBA/DCOM/JavaBeans/...:
Components Off-The-Shelf (COTS)

- Components may be nested
- Dynamic call in CORBA

Component Model in Architecture Systems

- Ports: abstract interface points (as in Linda)
  - in(data), out(data)
  - Components may be nested

- Connectors as special communication components
### Web Services

- Binding procedure is interpreted, not compiled
- More flexible:
  - When interface changes, no recompilation and rebinding
  - Ubiquitous http protocol – independent of a specific ORB

#### Architecture Systems

- Unicon, ACME, Darwin
  - feature an Architecture Description Language (ADL)
- Split an application into:
  - Application-specific part (encapsulated in components)
  - Architecture and communication (in architectural description in ADL)
  - Better reuse since both dimensions can be varied independently

### Web Services as Composition System

- Component Mode
  - Content: not important
  - Binding points are described by XML
  - Binding procedure is interpretation of SOAP
  - Secrets: distribution, implementation language

- Composition Technique
  - Adaptation for distributed systems (marshalling) and mixed-language systems
  - Glue: SOAP, http

- WSDL, UDDI, BPEL

### Architecture can be exchanged independently of components

- Reuse of components and architectures is fundamentally improved
The Composition Language: ADL

- Architectural description language, ADL
  - ADL-compiler
  - XML-Readers/Writers for ADL
  - XADL is a new standard exchange language for ADL based on XML

- Graphic editing of systems
- Checking, analysing, simulating systems
  - Dummy tests
  - Deadlock checkers
  - Liveness checking

Architecture Systems as Composition Systems

- Source or binary components
- Binding points: ports
- Adaptation and glue code by connectors
- Scaling by exchange of connectors

Architectural language (ADL)

Composition Language

Component Model
- Source or binary components
- Binding points: ports

Composition Technique
- Adaptation and glue code by connectors
- Scaling by exchange of connectors

Aspect Systems
- View Systems
- Software Composition Systems

Composition Language
- Operators
- Invasive Composition Language

Composition Filters
- Hyperslices

Architectural Systems
- Architecture as Aspect

Domain

Classical Component Systems
- Standard Components
- .NET CORBA
- Beans
- EJB

Object-Oriented Systems
- Objects as Run-Time Components
- C++
- Java

Modular Systems
- Modules as Compile-Time Components
- Ada-85
Graybox Component Models

Component integration
- Aspect oriented programming
- View-based composition

Aspects in Architecture

Aspects in Software

Aspect Systems

- Aspect languages
  - Every aspect in a separate language
  - Domain specific
  - Weaver must be built (is a compiler, much effort)

- Script-based Weavers
  - The weaver interprets a specific script or aspect program
  - This introduces the aspect into the core
Aspect Weavers Distribute Advice Components over Core Components

- Aspects are crosscutting
- Hence, aspect functionality must be distributed over the core

Aspect Systems As Composition Systems

Component Model
- Core- and aspect components
- Aspects are relative and crosscutting

Composition Technique
- Adaptation and glue code by weaving
- Weaving is distribution

Composition Language

Composition Systems with composition operators and expressions

- Hyperspace Programming [Ossher et al., IBM]
- Piccola [Nierstrasz, et al., Berne]
- Metaclass composition [Forman/Danforth, Cointe]
- Invasive composition [Assmann]
- Formal calculi
  - Lambda-N calculus [Dami]
  - Pi-L calculus [Lumpe]
Connectors are Composition Operators

- Blackbox Composition
  - Client
  - Library

- Invasive Composition
  - Client
  - Library

Blackbox connection with glue code

Composers can be used for inheritance

- Extension can be used for inheritance (mixins)

  A mixin is a class (i.e., a set of features) by which a superclass can be extended to derive a subclass. The mixin class itself is final, i.e., cannot be subclassed.

- Mixin-based inheritance:
  - copy first superclass
  - extend with fragments of second superclass (mixin)

Composers Generalize Aspect Weavers in AOP

- Complex composers distribute aspect fragments over core fragments
  - Distributors extend the core
  - Distributors are more complex operators, defined from basic ones

Composition Languages

- Composition languages describe the structure of the system in-the-large ("programming in the large")

- Composition programs combine the basic composition operations of the composition language

- Composition languages can look quite different
  - Standard languages, such as Java
  - Makefiles

- Enables us to describe large systems

Composition program size: 1
System size: 10
Conclusions for Composition Systems

- Components have a composition interface
  - Composition interface is different from functional interface
  - The composition is running usually before the execution of the system
  - From the composition interface, the functional interface is derived
- System composition becomes a new step in system build

Steps in System Construction

- We need component models and composition systems on all levels of system construction

  System composition
  (System generation)

  System compilation

  System deployment

  System execution

Fragment Components Have Hooks

- Software variation points, hooks
  - Method entries/exits
  - Generic parameters

Invasive Composition

Invasive composition adapts and extends components at hooks by transformation
The Component Model of Invasive Composition

- The component is a fragment container (fragment box)
  - a set of fragments/tag elements
- Uniform representation of
  - a fragment
    - a class, a package, a method
  - a set of fragments
    - an aspect
    - a meta description
    - a composition program

Implicit Hooks In Software

- Given by the programming language
- Example: Method entry/exit

```
Method entry m () {
  abc...
  cde...
}
Method exit
```

Declared Hooks

Declared Hooks are declared by the component writer as code parameters

The Composition Technique of Invasive Composition

Invasive Composition adapts and extends components at hooks by transformation

A composer transforms unbound to bound hooks

composer: fragment box with hooks -> fragment box with bound hooks
**Method Entry**

```
m (){
    abc.
    cde.
}
```

**Method Exit**

```
m (){  
    print("enter m");
    abc.
    cde.
    print("exit m");
}
```

```
component.findHook("MethodEntry").extend("print("enter m");");
component.findHook("MethodExit").extend("print("exit m");");
```

**Generic Types**

```
<< ClassBox >>
class SimpleList {
    genericTType elem;
    SimpleList next;
    
    genericTType getNext() {
        return next.elem;
    }
}
```

```
<< ClassBox >>
class SimpleList {
    WorkPiece elem;
    SimpleList next;
    
    WorkPiece getNext() {
        return next.elem;
    }
}
```

**Generic Modifiers**

```
/* @hook Modifier MY */
{  
    public print() {
        System.out.println("Hello World");
    }
}
```

```
Component methodComponent = cs.createMethodBox();
Hook modif = methodComponent.findHook("MY");
if (parallelVersion) {  
    modif.bind("synchronized");
} else {  
    modif.bind(" ");
}

synchronized public print () {
    System.out.println("Hello World");
}
```

**Generic Statements**

```
// @hook Statement MY
{  
    public print() {
        System.out.println("Hello World");
    }
}
```

```
Component methodComponent = cs.createMethodBox();
Hook statement = methodComponent.findHook("MY");
if (StdoutVersion) {  
    statement.bind("System.out.println("Hello World");");
} else {  
    statement.bind("FileWriter.println("no way");");
}

public print () {
    System.out.println("Hello World");
}
```

```
Component methodComponent = cs.createMethodBox();
Hook statement = methodComponent.findHook("MY");
if (StdoutVersion) {  
    statement.bind("System.out.println("Hello World");");
} else {  
    statement.bind("FileWriter.println("no way");");
}

public print () {
    FileWriter.println("no way");
}
```
The Composition Technique of Invasive Composition

- **Composer**
- Invasive hook transformation

Composition Operators

**Basic operators:**
- bind hook (parameterization)
- generalized generic program elements
- rename component, rename hook
- remove value from hook (unbind)
- extend
  - extend in different semantic versions

**Compound operators...**

Invasive Composition as Composition System

- **Component model**
  - Source or binary components
  - Greybox components
  - Composition interfaces with declared implicit hooks

- **Composition technique**
  - Controlled by composition programs
  - Algebra of composition operators (basic and compound operators)
  - Uniform on declared and implicit hooks

Standard Language (Java)

Composition language

The COMPOsition SysTem

**COMPOST**

- **Library of static meta-programs**
- Composition language Java
- Reifies concepts Components, Hooks, Composers

- **Uni Karlsruhe/Uni Linköping 1998-2003**
  - http://www.the-compost-system.org
  - Version 0.78 of 2003
  - Continued at TU Dresden since 2004

Unification of Development Techniques

With the uniform treatment of declared and implicit hooks, several technologies can be unified:
- Generic programming
- Inheritance-based programming
- Connector-based programming
- View-based programming
- Aspect-based programming

Summary:
Component-based Systems

- are produced by component systems or composition systems
- have a central relationship that is tree-like or reducible
- support a component model
- allow for component composition with composition operators
- and – in the large – with composition languages
- Historically, component models and composition techniques have been pretty different
  - from compile time to run time
- Blackbox composition supports variability and adaptation
- Graybox composition also supports extensibility